手続き型言語におけるアルゴリズミックデバッギングの一実現方式

下村 隆夫†

システムのガイドに従ってバグを究明する従来のアルゴリズミックデバッギング手法では，手続き型言語には適用できない，検出できるのはバグを含む関数であり，バグを含む文まで限定することはできない，文の記述漏れに関するバグは検出できない等の問題点があった．これに対して，手続き型言語を対象とし，変数値エラーに対して，変数値エラーを引き起こした可能性のある文の集合である Critical Slice を用いた決定性のバグ究明方式が提案されている．本論文では，プログラムのテストにおいて観察される，出力異常エラー(変数値エラーを含む)，出力漏れエラー，および，出力なし無限ループエラーの3つのエラーに対する Critical Slice に基づいたバグ究明方式を提案する．出力漏れエラーがあると，プログラマがエラーを誤認する可能性について述べ，そのような場合には，出力文の間の依存関係を導入することにより，エラーの誤認を回避することができることを示す．また，本バグ究明方式を実現するためのシステムの構成，処理内容，および，ユーザインタフェースについても述べる．

An Implementation of Algorithmic Debugging for Procedural Languages

TAKAO SHIMOMURA†

In the conventional algorithmic debugging methods that locate a bug under the guidance of a system, there are some problems such that they cannot be applied to procedural languages, can determine only a faulty function, that is, cannot locate a faulty statement, or cannot detect a bug concerning omitted statements. On the other hand, an deterministic bug-locating method for variable-value errors in procedural languages has been presented, which is based on a critical slice, that is, a set of statements that might have caused an variable-value error. This paper presents a critical slice-based bug-locating method for three kinds of errors : wrong-output errors (including variable-value errors), missing-output errors, and no-output infinite-loop errors. It refers to cases in which a programmer may misidentify an error, and shows that the method this paper presents can avoid such misidentification of errors by introducing dependency between output statements. It also describes the system configuration, man-machine interfaces and behaviors of a system to implement this method.

1. はじめに

システムのガイドに従ってバグを究明するアルゴリズミックデバッギングには，Shapiro[1]，PRESET[2]，GADT[3),4]，PELAS[5)~7)]等がある．Shapiro, PRESET では関数型/論理型言語を対象とし，プログラマはシステムから提示された関数の正誤を，入出力パラメータの値を基に判定する．これを繰り返しながら，次第に誤りを含む部分を限定し，バグを含む関数を検出する．この方式では，検出できるのはバグを含む関数であり，バグを含む文まで限定することはできない．また，副作用のある手続き型言語には適用することができない．GADT は，この方式を手続き型言語にも適用しようという試みである．グローバル変数を参照するためのパラメータを関数に追加し，副作用のない同値な関数型プログラムに変換してからバグの究明を行う．また，Static Slicing[8]を利用することにより，値の誤っている出力パラメータに関係する関数を特定している．しかし，グローバル変数をパラメータで渡すようにプログラムを変換しても，グローバル変数の参照漏れや設定漏れは検出できないという問題が残る．また，この方式でも，検出できるのはバグを含む関数であり，バグを含む文まで限定することはできない．一方，PELAS では，手続き型言語を対象とし，実行した文の間の依存関係を実行順とは逆向きに順に調べ

† ATR 通信システム研究所
ATR Communication Systems Research Laboratories

ていくことにより，バグを含む文を限定することができる．しかし，文の記述漏れ等のバグは検出できないという問題がある[8]．

筆者は，手続き型言語を対象とし，変数値エラーに対して，Critical Slice を用いた決定性のバグ究明方式を提案した[9]．Critical Slice はエラーを引き起こす可能性のある文からなる集合であり，この Critical Slice を分割しフローデータの値の正誤を判定することにより，文の記述漏れを含む，任意のバグを究明することができる．

本論文では，プログラムのテストにおいて観察される，出力異常エラー（変数値エラーを含む），出力漏れエラー，および，出力なし無限ループエラーの3つのエラーに対する Critical Slice に基づいたバグ究明方式を提案する．出力漏れエラーがあると，プログラマがエラーを誤認する可能性について述べ，そのような場合でも，出力文の間の依存関係を導入することにより，エラーの誤認を回避することができることを示す．また，本バグ究明方式を実現するためのシステムの構成，処理内容，および，ユーザインタフェースについても述べる．

2. Critical Slice の概要

まず，文献9)で提案された Critical Slice について，概説する．

2.1 バグの分類

プログラムの文に関するバグは，次の4つ，あるいは，これらの組み合わせからなる．

（1） 文記述漏れバグ
　文の記述が漏れている場合．
（2） 文記述過多バグ
　余分な文の記述がある場合．
（3） 文記述誤りバグ
　（a） 代入文において，右辺の式の記述を誤った場場合，
　（b） 分岐文，ループ文において，条件式の記述を誤った場合，
　（c） 出力文において出力アーギュメントの記述を誤った場合．
（4） 名前記述誤りバグ
　（a） 代入文において，左辺の変数の名前を誤った場合，
　（b） 入力文において入力変数の名前を誤った場合．

本論文では，文記述誤りバグと文記述過多バグを総称して**値誤りバグ**，文記述漏れバグと名前記述誤りバグを総称して**設定漏れバグ**と呼ぶこととする．

2.2 Critical Slice の定義

ある入力データを与えてプログラムを実行した場合，実行されたパス（命令の列）を**実行系列**と呼ぶ．t 番目に命令の実行が行われた時点を実行時点 t と呼ぶ．実行時点 t に関して，以下の記号を定義する．

$\mathrm{Ins}(t)$　t 番目に実行された命令．
$\mathrm{Use}(t)$　実行時点 t における命令 $\mathrm{Ins}(t)$ の実行で使用された変数の集合．

実行時点の間に，以下に示す4つの依存関係 Def, Ctl, CtlDef, OmsCond を定義する．

（1） $\mathrm{Def}(t,v)$ (Definition)…実行時点 t より前で，変数 v に最後に値を設定した実行時点である．

（2） $\mathrm{Ctl}(t)$ (Control)…実行時点 t の実行の有無を決定する命令を実行した実行時点の集合である．

（3） $\mathrm{CtlDef}(t,v)$ (Control-Definition)…次のように定義される．

$$\mathrm{CtlDef}(t,v)=\mathrm{Ctl}(\mathrm{Def}(t,v))-\mathrm{Ctl}(t).$$

$\mathrm{CtlDef}(t,v)$ は，分岐命令あるいはループ命令の実行時点の集合であり，それらの命令の制御移行により変数 v の値を設定する命令が実行されることになり，かつ，そこで設定された値が実行時点 t で使用している変数 v の値となっているという性質をもつ．

（4） $\mathrm{OmsCond}(t,v)$ (Omission-Conditional)…次のように定義される．

$\mathrm{OmsCond}(t,v)=\{$実行時点 $j \mid \mathrm{Def}(t,v)<j<t,\ j\notin \mathrm{Ctl}(t)$, かつ, $\mathrm{Ins}(j)$ は分岐命令あるいはループ命令であり，その制御移行が変われば，その分岐文あるいはループ文内で変数 v を定義する可能性がある$\}$．

$\mathrm{OmsCond}(t,v)$ は，分岐命令あるいはループ命令の実行時点の集合であり，それらの命令の制御移行が変われば，変数 v の値を設定する可能性があり，かつ，そのために，それらの命令の制御移行が変われば，実行時点 t で使用している変数 v の値を変えたかもしれないという性質をもつ．

（5） Critical 実行時点集合 $\mathrm{CriticalEP}(t,v)$ の定義

後で述べる出力異常エラーに対するバグ究明方式（3.2節参照）にも適用できるようにするため，本論文では，実行時点 t における，変数 v に関する Critical 実行時点集合 $\mathrm{CriticalEP}(t,v)$ を，関数 CEP を用いて，次のように定義する．

```
function CEP(i : 実行時点; AffectUse : 変数の集合; e : 実行時点) return 実行時点の集合 is
   X, EP : 実行時点の集合;
begin
   if i = e then
      X := Def(i, AffectUse) ∪ OmsCond(i, AffectUse) ∪ CtlDef(i, AffectUse) ∪ Ctl(i);
   else
      X := Def(i, AffectUse) ∪ OmsCond(i, AffectUse) ∪ CtlDef(i, AffectUse);
   end if;
   if X = φ then
      return (φ);
   end if;
   EP := X;
   for each x∈X loop
      EP := EP ∪CEP(x, Use(x), e);
   end loop;
   return EP;
end CEP;
```

図 1 関数 CEP
Fig. 1 Function CEP.

$\mathrm{CriticalEP}(t, v) = \mathrm{CEP}(t, \{v\}, t)$.

実行時点 i, e $(i \leqq e)$, 変数の集合 AffectUse に対して, 関数 CEP$(i$, AffectUse, $e)$ は, **図 1** に示すように再帰的に定義される. 関数 CEP において, Ctl(t) となる実行時点を CriticalEP(t, v) に含めているのは, 実行時点 t において変数 v の値が誤っている変数値エラーにおいては, 1)実行時点 t における制御フローは正しい (すなわち, Ctl(t) 内の実行時点における制御移行結果が正しい) が変数 v の値が誤っている場合と, 2)実行時点 t における制御フロー自体が既に誤っている場合とがあるためである.

Critical 実行時点集合の要素を Critical 実行時点と呼ぶ. Critical 実行時点をノード, Critical 実行時点の間の Def, Ctl, CtlDef, OmsCond 関係をアークとすることにより定義されるグラフを **Critical-Flow グラフ**と呼ぶ (図 3 参照).

(6) CriticalSlice(t, v) の定義

実行時点 t における, 変数 v に関する Critical Slice, CriticalSlice(t, v) を以下のように定義する.

$\mathrm{CriticalSlice}(t, v) = \{\mathrm{Ins}(j) \mid j \in \mathrm{CriticalEP}(t, v)\}$.

CriticalSlice(t, v) は, 実行時点の集合 CriticalEP (t, v) において実行された命令の集合である. Critical Slice は命令の集合であるが, Static Slic[10)~14)]や Dynamic Slice[15)~17)]のように実行可能な部分プログラムとなることを意図していない.

2.3 Critical Slice の性質

(1) 値誤りバグ潜在域

変数値エラーに対して, プログラム内の命令の集合 Xで, X以外の命令については, いずれの命令に値誤りバグがあっても, その変数値エラーを引き起こすことがない場合, 集合Xをその変数値エラーに関する**値誤りバグ潜在域**と呼ぶ. 実行時点 t において, 変数 v の値が誤っている変数値エラーでは, 実行時点 t における変数 v に関する Critical Slice が, その変数値エラーに関する値誤りバグ潜在域となる.

(例) 2つの値 x, y の min, max を求めるプログラム min_max を **図 2** に示す. このプログラムに入力 $x=3$, $y=2$ を与えて実行させた時の実行系列を**図 3** に示す. 図 3 において, js は実行時点 j において命令 S が実行されたことを表す. この時, 実行時点 6 において変数 min の値 3 が誤っている (正しくは, 2). CriticalEP(6, min)= {1, 3, 4} より, CriticalSlice(6, min)= {Ins(1), Ins(3), Ins(4)} = {1, 3, 4} である. この例から分かるように, Critical-

```
1  get(x, y);
2  max := x;
3  min := x;
4  if x > y then    (x < y が正しい)
5     max := y;
   else
6     min := y;
   end if;
7  put(min, max);
```

図 2 プログラム min_max
Fig. 2 Program min_max.

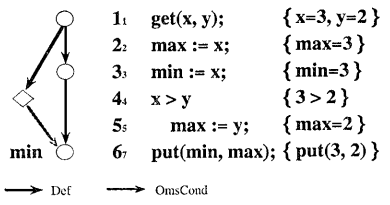


図 3 プログラム min_max の実行系列
Fig. 3 Execution sequence of program min_max.

```
1  get(m, n);
2  c := m;
3  d := n;
4  while d ≠ 0 loop
5     r := c / d;          (r := c mod d; が正しい)
6     c := d;
7     d := r;
   end loop;
8  x := c;
9  put(x);
```

図 4 プログラム gcd
Fig. 4 Program gcd.

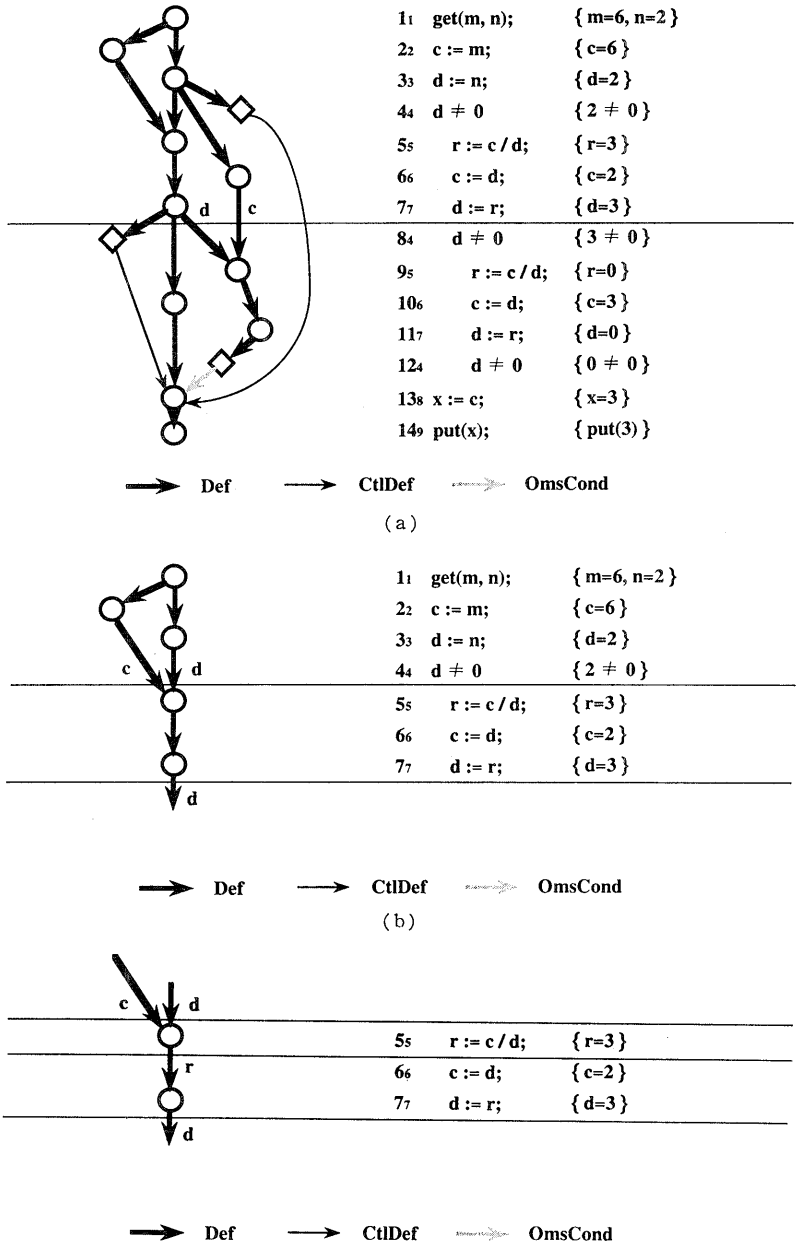


図 5 プログラム gcd のバグ究明
Fig. 5 Fault localization of program gcd.

Slice 内の命令 1, 3, 4 に値誤りバグがあると，変数 min に誤った値を生成してしまう可能性がある．一方，CriticalSlice 以外の命令 2, 5, 6 に値誤りバグがあっても，変数 min には同じ値が生成されることが分かる．

（2） フローデータ

Critical 実行時点集合 CriticalEP(t, v) に対して，実行時点 i ($1 \leqq i \leqq t$) における，**フローデータ集合 FlowData(i)** を次のように定義する．

FlowData(i)＝{変数 $x \mid r < i \leqq s$ となる．ある r, s

$\in$CriticalEP(t, v) が存在して，$x \in$Use(s)，$r =$Def(s, x)}．

FlowData(i) に含まれる変数を**フローデータ**と呼ぶ．$x \in$FlowData(i) とは，直感的には，実行時点 i より前に存在する，ある Critical 実行時点で変数 x に値が設定され，実行時点 i 以後に存在する，ある Critical 実行時点で，その値が参照されていることを意味する．

実行時点 t において変数 v に関する変数値エラーが発生しているとする．実行時点 i $(1 \leqq i \leqq t)$ の直前における制御フローが正しい（すなわち，Ctl(i) 内の各実行時点における制御移行が正しい）場合，次のことが成立する．

分割点 i の直前において，

(a) FlowData(i) 内のある変数の値が誤っていれば，実行時点 i より前にバグが存在する．

(b) FlowData(i) 内のすべての変数の値が正しければ，実行時点 i 以後にバグが存在する．

従って，制御フローの正しいある実行時点でCritical 実行時点集合を分割し，その分割点におけるフローデータの値の正誤を判定すること（これを分割検証と呼ぶ）により，バグの存在範囲を限定することができる．

(例) 2つの値 m, n の最大公約数を求めるプログラム gcd を**図4**に示す．プログラム gcd に入力 $m=6$, $n=2$ を与えて実行した時の実行系列を**図5**(a)に示す．値3が出力され，出力文の実行時における変数 x の値3が誤っている（正しくは，$x=2$）．出力文の実行時点 14 における変数 x に関する Critical-Flow グラフを図5(a)に示す．バグは以下の手順で見つけることができる．分割点は，説明の都合上，必ずしも，最適な分割点とはなっていない．

1) 実行時点8で分割する．まず，制御フローについて調べる．Ctl(8)={4} であり，実行時点4におけるループ命令 "$d \neq 0$" の制御移行は正しい．そこで，次に，フローデータ $c=2$, $d=3$ の値の正誤を判定する（図5(a)）．ここで，フローデータでない変数 m, n, r については，エラーに関係しないため，それらの値を調べる必要はない．

変数 d の値3が誤っている（正しい値は，$d=0$）．

2) 実行時点5で分割し，フローデータ $c=6$, $d=2$ の値の正誤を判定する（図5(b)）．

フローデータの値はすべて正しい．

3) 実行時点6で分割し，フローデータ $r=3$ の値の正誤を判定する（図5(c)）．

変数 r の値3が誤っている（正しい値は，$r=0$）．代入文5で定義した変数 r の値3は誤っており，代入文5で使用した変数 c, d の値は正しいことから，代入文5 ("$r := c/d$;") の文記述誤りバグが検出される．

3. バグ究明方式

3.1 出力に関するエラー

プログラムのテストにおいては，プログラムを実行した時の出力内容を観察することにより，誤り（以下，エラーと呼ぶ）を発見する．ここでは，プログラムのテストにおいて観察される，出力に関する次の3つのエラーに対するバグ究明方式について述べる．

(1) 出力異常エラー
出力内容の誤り，あるいは，余分な出力がある場合．この場合には，出力を実行した出力文を調べることにより，(a)値の誤っている出力変数がある場合（変数値エラー）と，(b)値の誤っている出力変数がない場合とに分かれる．同じ値を出力しながら，プログラムの実行が無限ループしている場合には，最初の異常な出力に関する出力異常エラーとして観察される．

(2) 出力漏れエラー
出力の漏れがある場合．

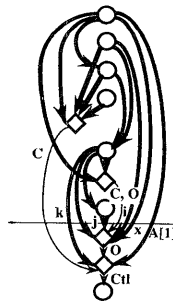
(3) 出力なし無限ループエラー
何の出力もなく実行が継続している場合．
このエラーが観察された場合，3.3節で述べるバグ究明方式を適用していくと，プログラムが実際には無限ループしていないことが分かる場合もありうる．

3.2 出力異常エラーに対するバグ究明

出力異常エラーを引き起こした出力文を実行した実

```
1  get(x, n, A);
2  i := 1;
3  j := n;
4  k := 1;
5  while A[k] ≠ x and i ≦ j loop
6      k := (i + j) / 2;
7      if x < A[k] then        (x > A[k] が正しい)
8          i := k + 1;
       else
9          j := k - 1;
       end if;
   end loop;
10 if A[k] = x then
11     put(k);
   else
12     put("Not found.");
   end if;
```

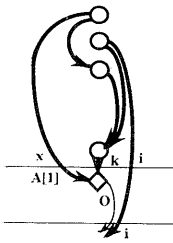
図6 プログラム binary_search1
Fig. 6 Program binary_search1.



1_1	get(x, n, A);	{x=6, n=2, A=(2,6)}
2_2	i := 1;	{i=1}
3_3	j := n;	{j=2}
4_4	k := 1;	{k=1}
5_5	A[k] ≠ x and i ≦ j	{2≠6 and 1≦2}
6_6	k := (i + j) / 2;	{k=1}
7_7	x < A[k]	{6<2}
8_9	j := k - 1;	{j=0}
9_5	A[k] ≠ x and i ≦ j	{2≠6 and 1≦0}
10_{10}	A[k] = x	{2=6}
11_{12}	put("Not found.");	{put("Not found.")}

→ Def ⟶ Ctl, C(CtlDef), O(OmsCond)

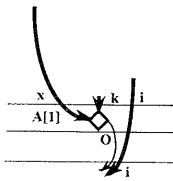
(a)



1_1	get(x, n, A);	{x=6, n=2, A=(2,6)}
2_2	i := 1;	{i=1}
3_3	j := n;	{j=2}
4_4	k := 1;	{k=1}
5_5	A[k] ≠ x and i ≦ j	{2≠6 and 1≦2}
6_6	k := (i + j) / 2;	{k=1}
7_7	x < A[k]	{6<2}
8_9	j := k - 1;	{j=0}

→ Def ⟶ Ctl, C(CtlDef), O(OmsCond)

(b)



7_7	x < A[k]	{6<2}
8_9	j := k - 1;	{j=0}

→ Def ⟶ Ctl, C(CtlDef), O(OmsCond)

(c)

図 7 プログラム binary_search1 のバグ究明
Fig. 7 Fault localization of program binary_search1.

```
1   get(x, n, A);
2   i := 1;
3   j := n;
4   k := 1;
5   while A[k] ≠ x and i ≦ j loop
6       k := (i + j) / 2;
7       if x > A[k] then
8           i := k;          (i := k + 1; が正しい)
          else
9           j := k - 1;
        end if;
    end loop;
10  if A[k] = x then
11      put(k);
    else
12      put("Not found.");
    end if;
```

図 8 プログラム binary_search2
Fig. 8 Program binary_search2.

行時点を t とする．値の誤っている出力変数を v，ErrVar＝{v} とする．出力文に制御が移行したこと自体が誤りであり，値の誤っている出力変数が存在しない場合には，ErrVar＝ϕ とする．CEP(t, ErrVar, t) によって求められる Critical 実行時点に対して，分割検証を行い，バグを究明すればよい．

(例) 与えられた値 x をもつ配列要素を捜し，その配列要素の添字の値を返すプログラム binary_search1 を**図 6** に示す．プログラム binary_search1 に入力 x＝6, n＝2, A＝(2, 6) を与えて実行した時の実行系列を**図 7** (a) に示す．"Not found." が出力され，値 6 をもつ配列要素が見つからない．この例では，値の誤っている変数の存在しない，出力異常エラーが発生している．

ErrVar＝ϕ であるから，CEP(11, ϕ, 11) によって求められる Critical 実行時点を分割検証する．実行時点 11 における，Critical-Flow グラフを図 7 (a) に示す．バグは以下の手順で見つけることができる．

1) 実行時点 9 で分割し，フローデータ $i, j, x, k, A[1]$ の値の正誤を判定する（図 7 (a)）．

変数 i の値 1，および，変数 j の値 0 が誤っている（この時点における正しい値は，i＝2, j＝2 である）．変数 i に着目して（詳細は割愛するが，この選択が最適である），バグを究明してみる．

2) 実行時点 7 で分割し，フローデータ i, x, k の値の正誤を判定する（図 7 (b)）．

フローデータの値はすべて正しい．

3) 実行時点 8 で分割し，実行時点 7 における分岐命令の制御移行結果の正誤を判定する（図 7 (c)）．

分岐命令の制御移行結果が誤っており，その分岐命令で使用した変数の値はすべて正しいことから，分岐命令 7 ("x<$A[k]$") の文記述誤りバグである．

3.3 出力なし無限ループエラーに対するバグ究明

出力なし無限ループエラーが観察される場合には，プログラムの実行を強制中断し，強制中断した時点 i を分割点と考え，制御フローの検証 (Ctl(i)) に含まれる実行時点における制御移行結果の正誤を順に判定す

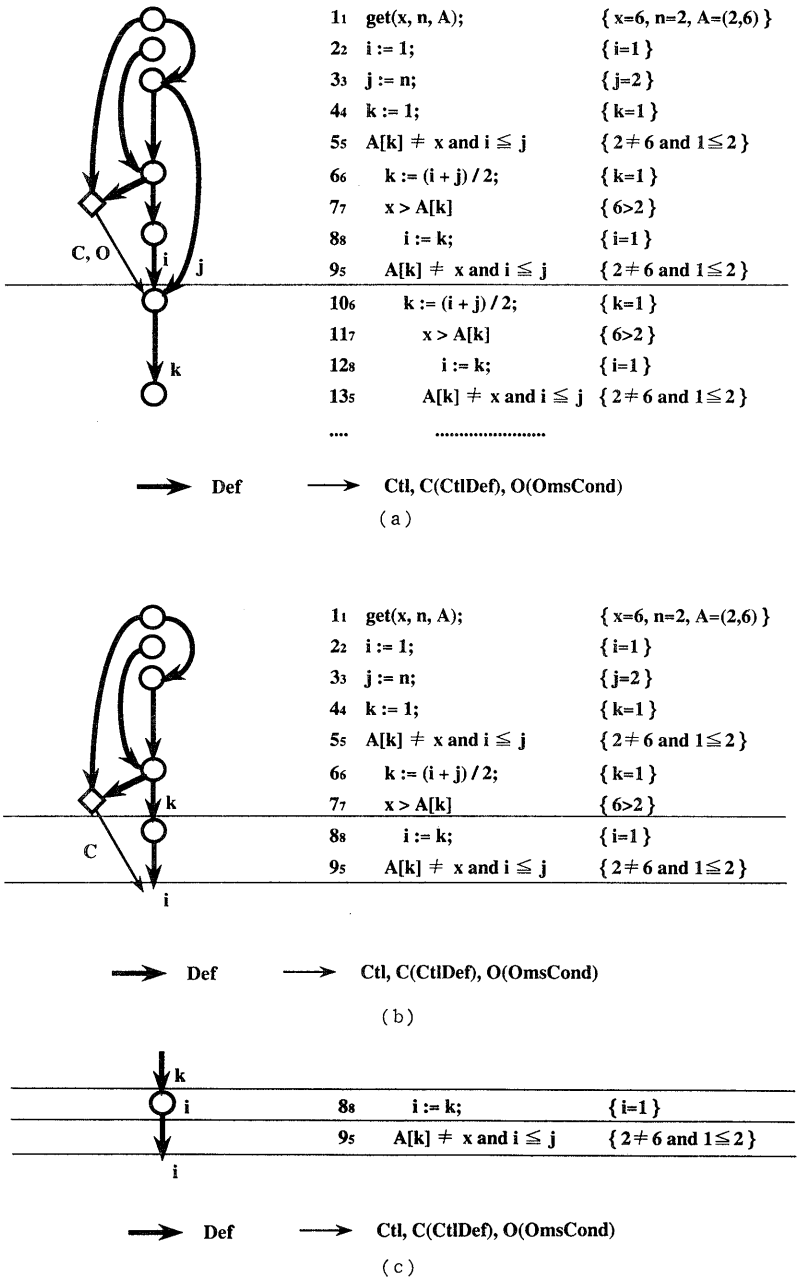


図 9 プログラム binary_search2 のバグ究明
Fig. 9 Fault localization of program binary_search2.

ること）を行う．

分割点 i を含むループ文については，そのループ文内を繰り返し実行すべき回数を n とすると，$n+1$ 回以上，繰り返されている場合には，$n+1$ 回目に実行されたループ命令の制御移行結果の誤りである．そのループ命令を実行した実行時点を t，そのループ命令で使用した変数で値の誤っているものを v とすると，CEP(t, $\{v\}$, t) によって求められる Critical 実行時点に対して分割検証を行い，バグを究明すればよい．ループ命令で使用した変数で値の誤っているものが存在し

なければ，そのループ命令の文記述誤りバグである．

ループ文内の繰り返し回数が n 以下である場合には，最後に実行されたループ文の繰り返し部分の実行系列に対して，制御フローの検証を続ける．この場合には，このループ文の中に入れ子になっている別のループ文が無限ループしている可能性がある．また，制御フローの検証の結果，Ctl(i) に含まれるすべての実行時点における制御移行結果が正しいと判定された場合には，プログラムは実際には無限ループしていなかったことになる．

(例) binary_search1 とは別の箇所にバグを含むプログラム binary-search2 を**図8**に示す．プログラム binary_search2 に入力 $x=6$, $n=2$, $A=(2,6)$ を与えて実行した時の実行系列を**図9**(a)に示す．プログラム binary_search2 は，出力なしで，無限ループを実行する．

実行を強制中断し，実行系列の検証を行う．ループ文5は，ループ文内を2回だけ実行して，ループから出なければならない．従って，3回目に実行されたループ命令の制御移行結果の誤りである．そのループ命令で使用した変数で値の誤っているものは，変数 $k=1$ と変数 $i=1$ である（正しくは，$k=2$, $i=2$）．ここでは，変数 k に着目して（この選択が最適である），バグを究明する．実行時点13における変数 k に関する Critical-Flow グラフを図9(a)に示す．バグは以下の手順で見つけることができる．

1) 実行時点10で分割し，フローデータ i, j の値の正誤を判定する（図9(a)）．

変数 i の値1が誤っている（正しい値は2）．

2) 実行時点8で分割し，フローデータ k の値の正誤を判定する（図9(b)）．

フローデータ k の値1は正しい．

3) 実行時点9で分割し，フローデータ i の値の正誤を判定する（図9(c)）．

代入文8で定義した変数 i の値1は誤っており，代入文8で使用した変数 k の値は正しいことから，代入文8 ("$i:=k$;") の文記述誤りバグである．

3.4 エラー誤認の回避

ここでは，まず，出力漏れエラーがあると，プログラマがエラーを誤認する可能性について述べる．次に，そのような場合でも，出力文の間の依存関係を導入することにより，エラーの誤認を回避することができ，出力漏れエラーを含むすべてのエラーに対して，バグを検出できることを示す．

```
1  get(n);
2  if n≧1 then
3      put('A');
   end if;
4  if n≧2 then
5      put('B');
   end if;
6  if n≧3 then
7      put('C');
   end if;
```

(a) 正しいプログラム prog_out1
(a) Correct program prog_out1

```
1  get(n);
2  if n≦1 then          (n≧1 が正しい)
3      put('A');
   end if;
4  if n≧2 then
5      put('B');
   end if;
6  if n≦3 then          (n≧3 が正しい)
7      put('B');        (put('C'); が正しい)
   end if;
```

(b) バグを含んだプログラム prog_out2
(b) Faulty program prog_out2

```
CP : integer := 0;
procedure put(c: in character) is
begin
    CP := CP + 1;
    OUT[CP] := c;
end;
```

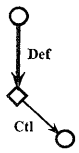
(c) 手続き put
(c) Procedure put

図10 エラー誤認の回避
Fig. 10 Avoiding misidentification of errors.

(1) エラーの誤認

図10(a)に示した正しいプログラム prog_out1 に入力 $n=2$ を与えて実行させると，出力は，"AB" となる．一方，図10(b)に示したバグを含んだプログラム prog_out2 に入力 $n=2$ を与えて実行させると，出力は，"BB" となる．プログラム prog_out2 の実行では，最初のBの出力の直前に，Aの出力が漏れていて，かつ，最初のBの出力の直後に，余分なBの出力がある．

しかし，プログラマは，プログラム prog_out2 の誤った実行結果を見て，2つのBの出力の中，2番目のBの出力は正しいが，最初のBの出力が誤っていると誤認する可能性がある．すなわち，実行時点4において，値の誤っている出力変数が存在しない出力異常エラーが発生していると認識した場合には，CEP $(4, \phi, 4)$ によって求められる Critical 実行時点に対して，分割検証を行うことになる（**図11**）．まず，実行時点3で分割すると，フローデータ n の値2は正し



1_1	get(n);	{n=2}
2_2	n≦1	{2≦1}
3_4	n≧2	{2≧2}
4_5	put('B');	{put('B')}

図 11 プログラム prog_out2 の Critical-Flow グラフ
Fig. 11 Critical-Flow graph of program prog_out2.

い．そこで，次に，実行時点4で分割すると，実行時点3における分岐命令（$n \geqq 2$）の制御移行結果も正しい．従って，出力異常エラーではなかったことが分かる．

（2） 出力依存関係の導入

プログラマがエラーを誤認する可能性がある場合には，出力文の間の依存関係を導入することにより，プログラマによるエラーの誤認を回避し，バグを的確に究明することができる．言い換えれば，プログラマがプログラム prog_out2 の実行結果を見て，最初のBの出力が誤っていると認識しても（これは自然な認識であるが），バグを見つけることができる．

出力文を実行すると，配列 OUT に出力された値が順に格納されるものとみなす．すなわち，手続き put では，図10(c)に示す処理を行うと考える．従って，プログラム prog_out2 では，**図12**(a)に示す処理を実行するとみなす．そして，プログラマが認識した，最初のBの出力に関する出力異常エラーは，出力文を実行した時点における配列要素 OUT[*CP*] の値に関する変数値エラー，すなわち，変数 *CP* と変数 *c* に関する変数値エラーとして扱う．出力異常エラーに対するバグ究明方式を適用し，CEP(6, {*CP*}, 6) によって求められる Critical 実行時点の分割検証を行うことにより，以下の手順でバグを見つけることができる．

1) 実行時点4で分割し，フローデータ $n(=2)$, *CP* (=0) の値の正誤を判定する（図12(a)）．

この時点では，既に出力が行われていなければならないため，*CP* の値0が誤っている．

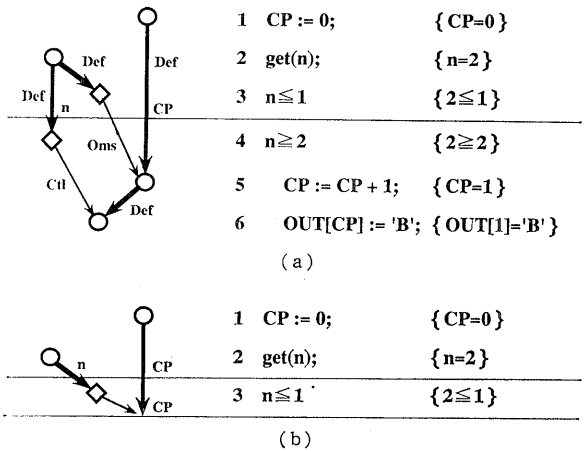
2) 実行時点3で分割し，フローデータ $n(=2)$, *CP*(=0) の値の正誤を判定する（図12(b)）．

フローデータ *n*, *CP* の値は正しい．

3) 実行時点3における分岐命令 ($n \leqq 1$) の制御移行結果の正誤を判定する．

分岐命令の制御移行結果は誤っており，その分岐命令で使用した変数の値はすべて正しいことから，分岐命令 ($n \leqq 1$) の文記述誤りバグである．

出力依存関係を導入した場合には，出力漏れは次の出力の出力異常エラーとして認識される．次の出力が存在しない場合には，プログラムの実行終了時点における変数 *CP* の値の変数値エラーとして認識すればよい．



1	CP := 0;	{CP=0}
2	get(n);	{n=2}
3	n≦1	{2≦1}
4	n≧2	{2≧2}
5	CP := CP + 1;	{CP=1}
6	OUT[CP] := 'B';	{OUT[1]='B'}

(a)

1	CP := 0;	{CP=0}
2	get(n);	{n=2}
3	n≦1	{2≦1}

(b)

図 12 プログラム prog_out2 のバグ究明
Fig. 12 Fault localization of program prog_out2.

4. 実 現 方 式

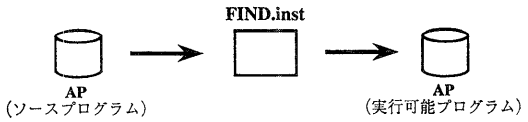
手続き型言語に対して，本論文で提案するアルゴリズミックデバッギングを実現するための方式について述べる．ここでは，SUN/UNIX ワークステーション上に試作中のデバッギングシステム FIND (Fault-locating INtelligent Debugger) を例にとり，システムの構成，ユーザインタフェース，および，処理内容について，具体的に記述する．FIND は C 言語で記述されたプログラムの出力異常エラーに対するバグを検出する．

4.1 処 理 形 態

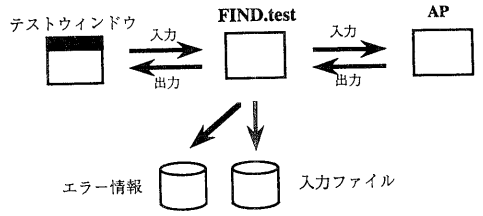
FIND におけるバグ究明処理は，次の3つのフェーズから構成される．

1) インストルメントフェーズ…デバッグ対象プログラム（以下，AP とする）に，次のテストフェーズ，および，デバッグフェーズにおいて AP の実行を制御するために必要なソースコードを埋め込んだ後，コンパイル，リンクを行い，実行可能なプログラムを生成する．

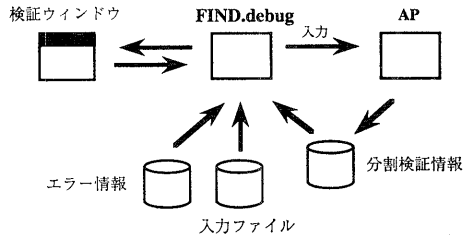
2) テストフェーズ…AP を実行してテストする．



(a) インストルメントフェーズ
(a) Instrumentation phase



(b) テストフェーズ
(b) Testing phase



(c) デバッグフェーズ
(c) Debugging phase

図 13 システム構成
Fig. 13 System configuration.

3) デバッグフェーズ…テストフェーズで発見された出力異常エラーに対して，バグを検出する．

システム構成を**図 13** に示す．

4.2 ユーザインタフェース

(1) テストフェーズ

テストウィンドウに AP への入力および AP からの出力が表示される．出力内容の中の誤っている部分をマウスで選択することにより，ユーザがエラーを指摘する．システムが選択された部分に対応する出力文，および，誤った出力を行った変数名，値を表示するので，ユーザはエラー内容を確認する．

(2) デバッグフェーズ

システムが提示した実行系列内の分割点に対して，ユーザが検証ウィンドウ上で制御フローやフローデータの値の判定を行う．これを繰り返すと，最後に，エラーを引き起こした原因となるバグをシステムがユーザに提示する．ユーザは，検証ウィンドウで検証中の命令に対応する箇所を，ソースプログラムや実行系列を表示したウィンドウ上で参照することができる．テストウィンドウおよび検証ウィンドウの表示例を**図 14** に示す．

4.3 テストフェーズの処理

1) FIND が fork(), execve() の実行により AP を起動する．
2) ユーザから入力された入力データをパイプを通して AP に送るとともに，入力データファイルに記録する．
3) AP からの出力はパイプを通して受け取り，テストウィンドウに表示する．この時，出力内容の表示位置と，その出力を行った出力文の実行時点，および，値を出力した変数との対応を記録しておく．
4) ユーザが出力内容の中の誤っている部分をマウスでクリックした時，出力内容の表示位置との対応関係を基にして，出力異常エラーが発生した実行時点，および，値の誤っている変数をエラー情報として記録し，デバッグフェーズに引き継ぐ．

4.4 デバッグフェーズの処理

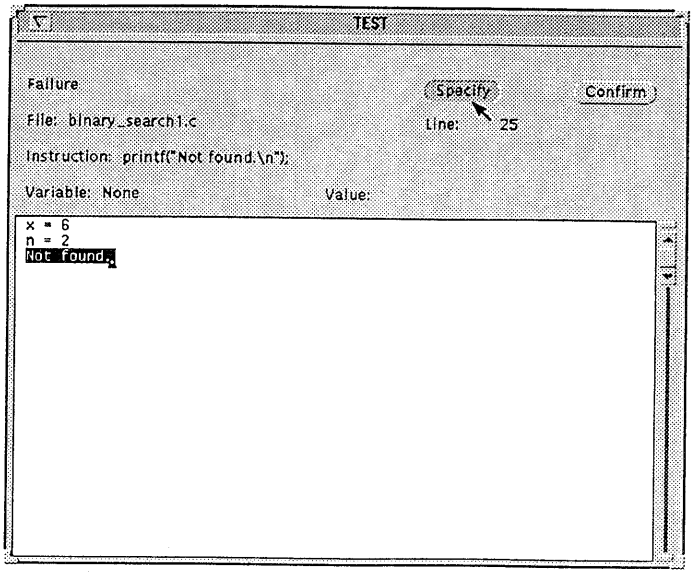
1) FIND が fork(), execve() の実行により AP を再起動する．
2) 入力データファイルに記録された入力データをパイプを通して AP に送り，AP を自動的に再実行する．
3) エラー情報を基に，AP を再実行しながら，実行系列内の分割点を決定し，分割点における制御フローやフローデータの値の判定に必要な情報を獲得する (4.5 節参照)．
4) 分割点における制御フローとフローデータの値を検証ウィンドウに表示し，ユーザにそれらの正誤を判定させる．
5) 以上を繰り返しながら，分割検証を進め，最後に，バグを検出する．

4.5 分割検証に必要な情報の獲得

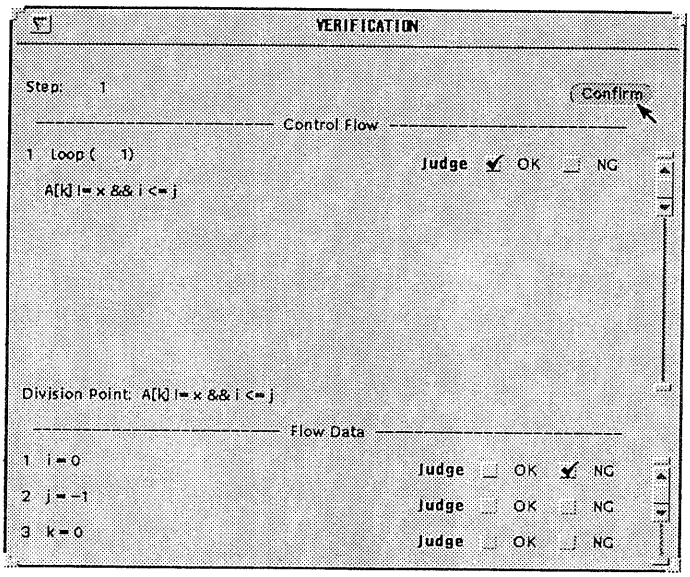
分割検証を行うためには，Critical 実行時点集合や Critical Slice を必ずしも求める必要はない．必要となるのは，分割点，および，分割点における制御フローとフローデータ (これらを分割検証情報と呼ぶ) である．以下では，AP を実行させながら，これらの分割検証情報を獲得する方法について述べる．

(1) AP 実行時に記録する情報

分岐命令あるいはループ命令を実行する場合には，その実行時点 p をスタック CurrentCtl にプッシュダ



（a） テストウィンドウ
（a） Test window



（b） 検証ウィンドウ
（b） Verification window

図 14 ユーザインタフェースの表示例
Fig. 14 User interface example.

ウンし，分岐文，ループ文の実行終了時には，実行時点をポップアップする．代入命令を実行し，変数wを定義する場合には，その実行時点dを LastDef(w) に設定し，実行時点dにおける CurrentCtl を LastDefCtl(w) に設定する．

また，分岐命令あるいはループ命令を実行する実行時点pにおいて，$w \in$ OmsVars(p) となる変数wに対して，実行時点pをOmsCondCandidates(w) に追加する．OmsVars(p) は，実行時点pにおける制御移行が変わると，定義される可能性のある変数の集合である．これは，分岐文の then 節，else 節，あるいは，ループ文内で定義される可能性のある変数の集合であり，分岐命令あるいはループ命令ごとに，インストルメントフェーズで求めておく．代入命令を実行し，変数wを定義する場合には，OmsCondCandidates(w) を空集合にする．

（2） 分割点の決定

ユーザが分割点で行う判定を容易にするため，検証対象範囲にある実行時点の数を2分する実行時点の近傍で，分岐命令やループ命令のネストレベルがなるべく小さいところを分割点として選択する．

（3） 分割点における制御フロー

分割点 i に対して，Ctl(i) に含まれる各実行時点pで実行された分岐命令あるいはループ命令の実行結果(True, False)，および，それらの命令で使用された変数の集合 Use(p) と，それらの変数の値を求める必要がある．このため，検証対象範囲内にある，分岐命令あるいはループ命令を実行する実行時点pでは，スタック CurrentCtl に，実行時点pのほかに，命令 Ins(p)，その実行結果，変数の集合 Use(p)，および，それらの変数の値をプッシュダウンする．Ctl(i) に関する情報は，分割点 i における CurrentCtl から求めることができる．

（4） 分割点におけるフローデータ

分割点 i におけるフローデータの変数名とその値を求める必要がある．まず，$i \leqq j$ となる各実行時点 j において，各変数 $w \in$ Use(j) に対して，次のように，Def(j, w)，CtlDef(j, w)，OmsCond(j, w) を求める．

Def(j, w) :=LastDef(w)；

CtlDef(j, w) :=LastDefCtl(w)−CurrentCtl;
OmsCond(j, w) :=OmsCondCandidates(w)
−CurrentCtl;

Def(j, w)<i ならば，変数 w が分割点 i におけるフローデータの候補となるため，変数wとその値を，実行時点 j に付随するリンク情報 LinkInf(j) に記録する．次に，各実行時点 k∈Def(j, w)∪CtlDef(j, w)∪OmsCond(j, w) に対して，i≦k ならば，実行時点 k に付随するリンク情報 LinkInf(k) を，実行時点 j に付随するリンク情報 LinkInf(j) に追加する．実行時点 t において，変数 v の値が誤っているという変数値エラーに対するバグを究明する場合には，分割点 i におけるフローデータに関する情報は，実行時点 t の変数 v に関するリンク情報から求めることができる．

5. おわりに

出力異常エラー，出力漏れエラー，および，出力なし無限ループエラーの3つのエラーに対するアルゴリズミックデバッギング方式について提案し，本方式を実現するためのシステムの構成，処理内容，および，ユーザインタフェースについて述べた．

出力異常エラーの発生している実行時点をユーザが直接，意識しなくてもすむようにするため，テストウィンドウ上でユーザがエラーを指摘する方式を採用した．また，テストフェーズですべてのテストをまとめ行い，エラーを発見するごとに，そのエラーに対するバグを究明するために必要となる分割検証情報を，バックグラウンドジョブとして別のプロセスで予め収集しておくと，デバッグフェーズでは，ユーザが直ちに分割検証を開始することができる．今後は，FIND のバグ究明効率，操作性に関して評価を行っていきたい．

謝辞 本研究に対して貴重な示唆と助言を頂きました National Institute of Standards and Technology の James R. Lyle 博士，および，Wayne State 大学の Bogdan Korel 博士に感謝いたします．

また，本研究を進めるにあたり日頃から励ましと助言を頂きました，NTT ソフトウェア研究所細谷僚一所長，後藤滋樹部長，伊藤正樹リーダに深謝いたします．

参考文献

1) Shapiro, E. Y.: *Algorithmic Program Debugging*, The MIT Press (1982).
2) Takahashi, H. and Shibayama, E.: PRESET—A Debugging Environment for Prolog, *Logic Programming Conference*, Tokyo, pp. 90-99 (1985).
3) Shahmehri, N., Kamkar, M. and Fritzson, P.: Semi-automatic Bug Localization in Software Maintenance, *Proceedings of Conference on Software Maintenance*, pp. 30-36 (Nov. 1990).
4) Fritzson, P., Gyimothy, T., Kamkar, M. and Shahmehri, N.: Generalized Algorithmic Debugging and Testing, *ACM SIGPLAN Notices*, Vol. 26, No. 6, pp. 317-326 (1991).
5) Korel, B. and Laski, J.: STAD—A System for Testing and Debugging: User Perspective, *Proceedings of Second Workshop on Software Testing, Verification, and Analysis*, pp. 13-20 (July 1988).
6) Korel, B.: PELAS—Program Error-Locating Assistant System, *IEEE Trans. Softw. Eng.*, Vol. 14, No. 9, pp. 1253-1260 (1988).
7) Korel, B. and Laski, J.: Algorithmic Software Fault Localization, *Proc. 24th Annual Hawaii International Conference on System Science*, pp. 246-252 (1991).
8) 下村隆夫：Program Slicing 技術とテスト，デバッグ，保守への応用，情報処理，Vol. 33, No. 9, pp. 1078-1086 (1992).
9) 下村隆夫：変数値エラーにおける Critical Slice に基づくバグ究明戦略，情報処理学会論文誌，Vol. 33, No. 4, pp. 501-511 (1992).
10) Weiser, M.: Programmers Use Slices When Debugging, *CACM*, Vol. 25, No. 7, pp. 446-452 (1982).
11) Weiser, M.: Program Slicing, *IEEE Trans. Softw. Eng.*, Vol. SE-10, No. 4, pp. 352-357 (1984).
12) Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs, *ACM Trans. Prog. Lang. Syst.*, Vol. 12, No. 1, pp. 26-60 (1990).
13) Weiser, M. and Lyle, J.: Experiments on Slicing-Based Debugging Aids, *Empirical Studies of Programmers*, pp. 187-197, Ablex Publishing Corporation (1986).
14) Lyle, J. R. and Weiser, M.: Automatic Program Bug Location by Program Slicing, *The Second International Conference on Computers and Applications*, pp. 877-883 (June 1987).
15) Korel, B. and Laski, J.: Dynamic Program Slicing, *Inf. Process. Lett.*, Vol. 29, No. 10, pp. 155-163 (Oct. 1988).
16) Korel, B. and Laski, J.: Dynamic Slicing of Computer Programs, *J. Systems Software*, Vol. 13, pp. 187-195 (1990).
17) Agrawal, H. and Horgan, J.: Dynamic Program Slicing, *ACM SIGPLAN Notices*, Vol. 25, No. 6, pp. 246-256 (1990).

(平成4年12月9日受付)
(平成6年9月6日採録)

下村　隆夫（正会員）

昭和24年生．昭和48年京都大学理学部数学科卒業．昭和50年東北大学大学院修士課程修了．同年日本電信電話公社（現 NTT）電気通信研究所勤務．昭和62年より NTT ソフトウェア研究所勤務．平成5年12月より ATR 通信システム研究所に出向中．平成4年4月から平成6年3月まで電気通信大学大学院情報システム学研究科客員助教授．工学博士．これまでに，汎用クロスアセンブラ，統計解析プログラム，仕様記述言語，グラフィックパッケージ，CASE ツール，テストカバレッジアナライザ，ビジュアルデバッガ，情報通信システムセキュリティの研究実用化に従事．ソフトウェアの設計，テスト，デバッグの自動化に興味をもつ．電子情報通信学会，ACM 各会員．